Announcement

• Homework #2 due next Tuesday!
• No office hour of Hung-Wei this Friday
  • Check the calendar of TA/tutors’ office hours
  • They will help you on assignments and quizzes
Recap: Basic steps of execution

- Instruction fetch: where? instruction memory
- Decode:
  - What’s the instruction?
  - Where are the operands? registers
- Execute ALUs
- Memory access
  - Where is my data? data memory
- Write back
  - Where to put the result
- Determine the next PC
Recap: Single-cycle processor

- The cycle time is determined by the longest instruction
- Could be very long, thinking about fetch data from DRAM
- Hardware is mostly idle
Fallacy about store/“sw”

- `sw $1, 0($2) : MEM[$2+0] = $1`
- Store does not have “destination register”
- Store does not change any register states
- Store does not do anything in “WB”
- Store uses “rs” and “offset” for ALU inputs
- The “rt” in store is a “source” of writing to memory
- Review your quiz #4.
Outline

• Pipelining
• Designing a 5-stage pipeline processor for MIPS
Pipelining

- Break up the logic with “pipeline registers” into pipeline stages
- Each stage can act on different instruction/data
- States/Control signals of instructions are held in pipeline registers
After the 5th cycle, the processor can do 5 instructions in parallel.
The processor can complete 1 instruction each cycle

CPI == 1 if everything works perfectly!
Single-cycle v.s. pipeline v.s.
The following diagram shows the latency in each part of a single-cycle processor:

If we can make each part as a “pipeline stage”, what’s the maximum speedup we can achieve? (choose the closest one)

A. 3.33
B. 4
C. 5
D. 6.67
E. 10

Speedup = \[
\frac{\text{\# of ins} * 1 * 10\text{ns}}{\text{\# of ins} * 1 * 3\text{ns}}
\]
Cycle time of a pipeline processor

- Critical path is the longest possible delay between two registers in a design.
- The critical path sets the cycle time, since the cycle time must be long enough for a signal to traverse the critical path.
- Lengthening or shortening non-critical paths does not change performance.
- Ideally, all paths are about the same length.
Limitations of pipelining

How many of the following descriptions about pipelining is correct?

- You can always divide stages into short stages with latches
- Pipeline registers incur overhead for each pipeline stage
- The latency of executing an instruction in a pipeline processor is longer than a single-cycle processor
- The throughput of a pipeline processor is usually better than a single-cycle processor
- Pipelining a stage can always improve cycle time

A. 1
B. 2
C. 3
D. 4
E. 5
Designing a 5-stage pipeline processor for MIPS
Basic steps of execution

- Instruction fetch: where? instruction memory
- Decode:
  - What’s the instruction?
  - Where are the operands? registers
- Execute ALUs
- Memory access
  - Where is my data? data memory
- Write back registers
  - Where to put the result
- Determine the next PC
Pipeline a MIPS processor

- Instruction Fetch
  - Read from instruction memory
- Decode
  - Figure out the incoming instruction?
  - Fetch the operands from the registers
- Execution
  - Perform ALU functions
- Memory access
  - Read/write data memory
- Write back results to registers
  - Write to the register file
From single-cycle to pipeline

Instruction Fetch

Instruction Decode

Execution

Memory Access

Write Back

Instruction Memory

Read Address

inst[31:0]

PCSrc = Branch & Zero

Control

Add

RegWrite

inst[31:25], inst[5:0]

Data Memory

Write Data

Zero

Add

Shift left 2

RegDst

RegWrite

ALUSrc

ALUop

Data Memory Access

Write Data

MemRead

MemWrite

Write Back

Will this work?
Pipelined processor

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
Pipelined processor

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw  $1, 0($12)
```
Pipelined processor

Where can I find these?

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
Pipelined processor

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9, $10, $11
sw $1, 0($12)
Pipelined processor

Is this right?

add $1, $2, $3
lw $4, 0($5)
sub $6, $7, $8
sub $9,$10,$11
sw $1, 0($12)
Pipelined processor

IF/ID

ID/EX

EX/MEM

MEM/WB

Instruction
Memory

Read Address
inst[31:0]

Add

inst[31:25],inst[5:0]

Control

RegWrite

File

Read Reg 1
Read Reg 2
Write Reg
Write Data

inst[25:21]

inst[20:16]

Control

RegWrite

Shift left 2

Add

ALUSrc

Zero

ALU

ALUop

RegDst

Data Memory

Address

Read Data

MemRead

Write Data

MemWrite

RegWrite

MemtoReg

m u x

m u x

m u x

m u x

m u x

m u x

m u x

m u x

m u x
5-stage pipelined processor
Simplified pipeline diagram

- Use symbols to represent the physical resources with the abbreviations for pipeline stages.
  - IF, ID, EXE, MEM, WB
- Horizontal axis represent the timeline, vertical axis for the instruction stream
- Example:

```plaintext
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9, $10, $11
sw  $1, 0($12)
```
Can we get the right result?

- Given the current 5-stage pipeline, how many of the following MIPS code can work correctly?

<table>
<thead>
<tr>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>b: lw $4, 0($1)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
</tr>
<tr>
<td>c: sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>bne $0, $7, L</td>
<td>sub $6, $7, $8</td>
</tr>
<tr>
<td>d: sub $9,$10,$11</td>
<td>sub $9, $1, $10</td>
<td>sub $9,$10,$11</td>
<td>sub $9,$10,$11</td>
</tr>
<tr>
<td>e: sw $1, 0($12)</td>
<td>sw $11, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
</tr>
</tbody>
</table>

- b cannot get $1 produced by a before WB
- both a and d are accessing $1 at 5th cycle
- We don’t know if d & e will be executed or not until c finishes

A. 0
B. 1
C. 2
D. 3
E. 4
Pipeline hazards
Pipeline hazards

• Even though we perfectly divide pipeline stages, it’s still hard to achieve CPI == 1.

• Pipeline hazards:
  • Structural hazard
    • The hardware does not allow two pipeline stages to work concurrently
  • Data hazard
    • A later instruction in a pipeline stage depends on the outcome of an earlier instruction in the pipeline
  • Control hazard
    • The processor is not clear about what’s the next instruction to fetch
Can we get the right result?

- Given the current 5-stage pipeline, how many of the following MIPS code can work correctly?

<table>
<thead>
<tr>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
<td>add $1, $2, $3</td>
</tr>
<tr>
<td>b: lw $4, 0($1)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
<td>lw $4, 0($5)</td>
</tr>
<tr>
<td>c: sub $6, $7, $8</td>
<td>sub $6, $7, $8</td>
<td>sub $0, $7, L</td>
<td>bne $0, $7, L</td>
</tr>
<tr>
<td>d: sub $9,$10,$11</td>
<td>sub $9, $1, $10</td>
<td>sub $9,$10,$11</td>
<td>sub $9,$10,$11</td>
</tr>
<tr>
<td>e: sw $1, 0($12)</td>
<td>sw $11, 0($12)</td>
<td>sw $1, 0($12)</td>
<td>sw $1, 0($12)</td>
</tr>
</tbody>
</table>

b cannot get $1 produced by a before WB
both a and d are accessing $1 at 5th cycle
We don’t know if d & e will be executed or not

Data hazard  Structural hazard  Control hazard
Structural hazard
**Structural hazard**

- What just happened here is problematic if we change one of the source register of the 2nd sub instruction?

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10, $1
sw $1, 0($12)
```

A. The register file is trying to read and write at the same cycle
B. The ALU and data memory are both active at the same cycle
C. A value is used before it’s produced
D. Both A and B
E. Both A and C
Structural hazard

- The hardware cannot support the combination of instructions that we want to execute at the same cycle.
- The original pipeline incurs structural hazard when two instructions compete the same register.
- Solution: write early, read late
  - Writes occur at the clock edge and complete long enough before the end of the clock cycle.
  - This leaves enough time for outputs to settle for reads.
  - The revised register file is the default one from now!

```
add $1, $2, $3
lw  $4, 0($5)
sub $6, $7, $8
sub $9,$10, $1
sw  $1, 0($12)
```